Concurrent Cache-Oblivious B-Trees Using Transactional Memory

Bradley C. Kuszmaul Jim Sukha

ABSTRACT

Cache-oblivious B-trees for data sets stored in external memory represent an application that can benefit from the use of transactional memory (TM), yet pose several challenges for existing TM implementations. Using TM, a programmer can modify a serial, in-memory cache-oblivious B-tree (CO B-tree) to support concurrent operations in a straightforward manner, by performing queries and updates as individual transactions. In this paper, we describe three obstacles that must be overcome, however, before one can implement an efficient external-memory concurrent CO B-tree.

First, CO B-trees must perform input/output (I/O) inside a transaction if the underlying data set is too large to fit in main memory. Many TM implementations, however, prohibit such transaction I/O. Second, a CO B-tree that operates on persistent data requires a TM system that supports durable transactions if the programmer wishes to be able to restore the data to a consistent state after a program crash. Finally, CO B-trees operations generate megalithic transactions, i.e., transactions that modify the entire data structure, because performance guarantees on CO B-trees are only amortized bounds. In most TM implementations, these transactions create a serial bottleneck because they conflict with all other concurrent transactions operating on the CO B-tree.

Of these three issues, we argue that a solution for the first two issues of transaction I/O and durability is to use a TM system that supports transactions on memory-mapped data. We demonstrate the feasibility of this approach by using LibXac, a library that supports memory-mapped transactions, to convert an existing serial implementation of a CO B-tree into a concurrent version with only a few hours of work. We believe this approach can be generalized, that memory-mapped transactions can be used for other applications that concurrently access data stored in external memory.

1. INTRODUCTION

Whereas most hardware and software transactional memory systems (e.g., [1, 2, 14–17, 19, 20, 23, 28]) implement atomicity, consistency, and isolation, but not durability (the so-called "ACID" properties [13]), we have developed a software transactional system that can provide full ACID properties for memory-mapped disk-resident data. This paper reports our experience using a transactional memory interface, with full ACID properties, to implement a cache-oblivous B-Tree.

Today, traditional B-trees [5, 10] are the dominant data structure for disk-resident data because they perform well in practice. In theory, traditional B-trees perform well in a performance model called the *Disk-Access Machine (DAM) Model* [4], an idealized two-level memory model in which all block transfers have unit cost, the block size is B, and the main-memory size is M. The choice of B defines the single granularity of optimization in the DAM model. For example, an optimized B-tree with fixed-sized keys has a branching factor of $\Theta(B)$, and thus requires $O(\log_B N)$ memory transfers for queries, which is optimal within the DAM model. The widespread use of B-trees suggests that the DAM model is used implicitly as a simplifying approximation for writing disk-intensive code.

It is difficult to choose the right value for B, however. The block size could be set to correspond to the CPU's cache line size (perhaps 64 bytes), to the disk's advertised block size (perhaps 4096 bytes), or possibly some larger value, such as the average track size on disk (on the order of $1/4$ megabyte for today's disks). Ideally, a B-tree would simultaneously minimize the number of cache lines, the number of disk blocks, and the number of tracks accessed during a query.

One way to avoid this block-size tuning problem is to employ data structures that work well no matter what the block size is. A ***cache-oblivious data structure*** is a data structure in which the parameters of the memory hierarchy (such as the cache-line size, the cache size, the disk block size, or the main memory size) are not coded explicitly, nor are they discovered by an active tuning process. In contrast, a ***cache aware*** data structure knows about the memory hierarchy.

Theoretical developments on cache-oblivious data structures and algorithms have shown in principal how to achieve nearly optimal locality of reference simultaneously at every granularity. In the *cache-oblivious model* [12, 26], an alternative to the DAM model, one can prove results about unknown memory hierarchies and exploit data locality at every scale. The main idea of the cache-oblivious model is that if it can be proved that some algorithm performs a nearly optimal number of memory transfers in a two-level model with unknown parameters, then the algorithm also performs a nearly optimal number of memory transfers on any unknown, multilevel memory hierarchy.

Our study focuses on the cache-oblivious B-tree (CO B-tree) data structure described in [6], and how to use a transactional memory interface to support concurrent queries and updates on the tree. Transactional memory is well-suited for programming a concurrent CO B-tree, since, arguably, a serial CO B-tree is already more complex to implement than a traditional serial B-tree. The CO B-tree is representative of the kind of data structure that we can implement with transactional memory: a data structure that may be more complicated, but asymptotically more efficient than the traditional alternative.

Furthermore, the cache-oblivious nature of the CO B-tree makes it difficult to parallelize the search tree operations using traditional methods for mutual exclusion. In a normal B-tree, the block size B presents a natural granularity for locking. For a CO B-tree that has no tunable parameters to set, however, the locking granularity would also need to be specified at an abstract level. Transactional memory interacts synergystically with cache-oblivious data structures because transactions allow the programmer to specify parallelism in an implementation-independent way.

A natural approach to programming a concurrent CO B-tree is to convert every query or update operation of a serial CO B-tree into its own transaction. We encountered three obstacles to making this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
WTW'06 June 10th, 2006, Ottawa, Ontario
Copyright 2006 Bradley C. Kuszmaul and Jim Sukha.

strategy work.

The first obstacle for the simple concurrent CO B-tree is transaction I/O. When the entire data set no longer fits into main memory, a query or update transaction may need to perform I/O to retrieve data from disk. If the programmer is working in a system with two levels of storage, then the programmer must make explicit I/O calls inside the transaction, typically through a buffer management subsystem, to bring a new page into memory and kick an existing page out. Buffer pool management adds another layer of complication to an already complex concurrent CO B-tree implementation. Furthermore, calls to the buffer management system should not be included as part of the transaction, since we can not easily undo I/O operations. Many proposed TM systems have not specified a programming interface or semantics for I/O operations that occur inside a transaction.

The second obstacle is transaction durability. For a CO B-tree that stores persistent data, the user would like the guarantee that the stored data will not be corrupted if the program accessing disk crashes. Database systems usually support durable transactions by updating a log on disk after every transaction commit. The log contains enough information to restore persistent data to a consistent state. Since TM systems already track the changes made by a transaction, support for durable transactions would be a natural extension.

The final obstacle with using TM on a CO B-tree is that the CO B-tree sometimes generates what we call *megalithic transactions*. A megalithic transaction is one that modifies a huge amount of state, effectively serializing performance. For the CO B-tree, there are some updates that must rebuild the entire data structure, producing a megalithic transaction. A megalithic transaction represents an extreme case because it runs for a long time and conflicts with all other transactions.

To address the first two obstacles of transaction I/O and transaction durability, we use LibXac, a page-based software transactional memory system that we developed, to implement the CO B-tree. LibXac supports transactions on memory-mapped data, allowing any application to concurrently access data from the same file on disk without explicit locking or I/O operations. We believe LibXac's interface is useful for generic concurrent external-memory data structures, since the issues of transaction I/O and durability are not specific to the CO B-tree.

More generally, external-memory data structures are a good match for a software transactional memory system (STM), because any runtime overheads of the STM can be amortized against the cost of disk I/O. Even if a memory access using an STM system costs an order of magnitude more than a normal memory access this overhead is small compared to the cost of a disk access for moderate-size transactions.

This paper describes the issues we encountered when using a transactional-memory interface to implement a concurrent cache-oblivious B-tree. Section 2 describes LibXac, our page-based software transactional memory implementation. Section 3 describes our experience implementing the CO B-tree, and explains how we used LibXac to address the first issues of transaction I/O and durability. Section 4 describes the CO B-tree structure in greater detail, explains how update operations can generate megalithic transactions, and discusses possible solutions for this problem. Finally, Section 5 concludes with a description of related work and directions for future work.

2. THE LibXac TM SYSTEM

We developed LibXac, a prototype page-based software transactional memory implementation that addresses the two problems of transaction I/O and durability. LibXac provides support for durable memory-mapped transactions, allowing programmers to write code that operates on persistent data as though it were stored in normal memory. In this section, we present LibXac's programming interface and an overview of its implementation.

Programming Interface

In many operating systems, different processes can share memory by using the system call `mmap` to memory-map the same file in shared mode. Programmers must still use locks or other synchronization primitives to eliminate data races, however, since this mechanism does not provide any concurrency control. Using LibXac, programmers transactionally memory-map a file using `xMmap`, and prevent data races by specifying transactions on the mapped region.

Figure 1 illustrates LibXac's basic features with two programs that access the same data concurrently. Both programs modify the first 4-byte integer in file `data.db`. The program on the left increments the integer, and the program on the right decrements it. When both programs run concurrently, the net effect is to leave the integer unchanged. Without some sort of concurrency control however, a race condition could cause the data to be corrupted.

Line 1 initializes LibXac, specifying the directory where LibXac will store its log files.[1] LibXac will write enough information to guarantee that the data in the file can be restored to a consistent state even if the program crashes during execution. Line 9 shuts down LibXac.

Line 2 opens a shared memory segment by using `xMmap()` to memory-map a particular file. The `xMmap()` function takes a filename and number of bytes to map as arguments, and returns a pointer to the beginning of the shared region. LibXac allows concurrency at a page-level granularity, and requires that specified length be a multiple of the page size.

Lines 3-8 contain the actual transaction. Line 5 is the actual body of the transaction.

Transactions are delimited by `xbegin()` and `xend()` function calls. The `xend()` function returns a status code that specifies whether the transaction was committed or aborted. If the transaction commits, then the `while` loop stops executing. Otherwise the code invokes the `backoff()` at Line 7, and then the `while` loop tries to run the transaction again. The application programmer can provide whatever implementation of the `backoff()` function they wish (for example, it might employ randomized exponential backoff [22]).

When using LibXac, the control flow for a program always proceeds through from `xbegin` to `xend`, even if the transaction is aborted. It is the programmer's responsibility to ensure that `xbegin()` and `xend()` function calls are properly paired, so that control flow does not jump out of the transaction without first executing `xend()`. LibXac also provides an `xValidate` function that the programmer can call in the middle of a transaction to check whether a transaction will need to abort because of a transaction conflict. The programmer can then insert code to stop executing a transaction that will not commit.

Nested transactions are automatically subsumed into the outermost transaction. This mechanism allows programmers to call functions inside a transaction that may themselves contain a transaction. An `xend()` call nested inside another transaction always succeeds, since the subtransaction successfully "commits" with respect to the outer transaction.

[1] LibXac can also be configured to support transactions that are not durable.

```
0  /* Increment the 1st integer in data.db */
1  xInit("/logs");
2  memptr = (int*)xMmap("data.db", 4096);
3  while (1) {
4    xbegin();
5    memptr[0] ++;
6    if (xend() == COMMITTED) break;
7    backoff();
8  }
9  xShutdown();
```

```
10 /* Decrement the 1st integer in data.db */
11 xInit("/logs");
12 memptr = (int*)xMmap("data.db", 4096);
13 while (1) {
14   xbegin();
15   memptr[0] --;
16   if (xend() == COMMITTED) break;
17   backoff();
18 }
19 xShutdown();
```

Figure 1: Two programs that access shared data concurrently, using LibXac transactions.

Memory Model

LibXac's memory model provides the standard transactional correctness condition, that transactions are ***serializable***. In fact, LibXac makes a stronger guarantee, that even transactions that end up aborting always see a consistent view of memory. Because LibXac employs a variation of a multiversion concurrency control algorithm [8], it can guarantee that an aborted transaction always sees a consistent view of the shared memory segment during execution, even if it conflicted with other transactions. Said differently, the only distinction between committed and aborted transactions is that a committed transaction atomically makes permanent changes that are visible to other transactions, whereas an aborted transaction atomically makes temporary changes that are never seen by other transactions.

When a transaction is aborted, only changes to the shared segment roll back however. Changes to local variables or other memory remain, allowing programmers to retain information between different attempts to execute a transaction. See [30] for more details LibXac's memory model.

Implementation

LibXac is implemented on Linux, without any modifications to the kernel or special operating system support. See [30] for a more thorough description of the implementation.

At a high-level LibXac executes transaction as follows.

1. When a transaction begins, the protection is set to "no access allowed" on the entire memory-mapped region.
2. The transaction's first attempt at reading (or writing) a page causes a segmentation fault on that page. The `SEGFAULT` handler installed by LibXac maps the appropriate version of the page into the transaction's address space as read-only data. LibXac relies on the ability to change the memory mapping of a particular virtual page, which Linux supports, but some operating systems may not.
3. When the transaction attempts to write to a page for the first time, the system must handle a second segmentation fault (because in Linux there is no easy way to distinguish between reads and writes.) LibXac generates a new version of the page and maps that version into the address space with read-write permissions.
4. When the `xend()` function is invoked, the runtime determines whether the transaction can commit. If so, then a log is generated of all the pages that were modified. That log includes a copy of both the old and new version of each page, as well as a commit record. The memory map is then reset to no-access.
5. Each transaction that modifies a page creates a new version of the page. Eventually, after the runtime determines that no transaction will ever need the version that is actually stored in the original file, it copies the youngest committed version of the page it can back to the original file. Thus, if the transaction processing becomes quiescent, the original file will contain the final version of the database.
6. After a committed version of a page is copied back into the original file, it eventually gets written to disk, either implicitly by the operating system's page eviction policy, or explicitly by a checkpoint operation.

During a transaction commit, our implementation performs the synchronous disk write required for logging, but we have not yet implemented checkpointing or a recovery program.

3. TRANSACTION I/O AND DURABILITY

In this section, we explain how transactional memory-mapping solves the issues of transaction I/O and durability, and describe our experience using LibXac to implement a CO B-tree.

A typical approach to managing disk-resident data is to program in a two-level store model with explicit disk I/O, managing a cache explicitly. Programming a two-level store is laborious, however, because the programmer must constantly translate from disk addresses to memory addresses. Moreover, programming cache-oblivious file structures using a two-level store is problematic, because the nature of cache-obliviousness precludes defining a particular block size for reading and writing.

We began with a serial implementation of a CO B-tree [18] that employs memory mapping, thereby handling the I/O issue automatically in the serial case. The Unix `mmap` function call provides the illusion of a single-level store to the programmer. The `mmap` function maps the disk-resident data into the virtual address space of the program, providing us with a large array of bytes into which we embed our data structure. Thus, the CO B-tree code can rely on the underlying operating system to handle I/O instead of calling explicit I/O operations.

The fact that LibXac's interface is based on memory-mapping also solves the problem of transaction I/O for a concurrent CO B-tree. The only additional complication concurrency introduces is the possibility of transactions with I/O aborting. Since LibXac only tracks operations on the mapped shared memory segment, however, the I/O operations are automatically excluded from the transaction. Thus, the application programmer does not need to worry about a transaction being aborted in the middle of an I/O operation.

LibXac also satisfies the requirement for transaction durability because it logs the contents of pages that a transactions modifies and synchronously writes the commit record to disk when a transaction commits. Our prototype system does not have recovery implemented, but it saves enough information to the log to allow a recovery program to restore the `xMmap`'ed file to a consistent state.

Because of the simplicity of LibXac's programming interface, starting with LibXac and a serial memory-mapped CO B-tree, we were able to easily create a CO B-tree that supports concurrent queries and updates with only a few hours of work. To demonstrate that our approach is practical, we ran a simple experiment performing 100,000 inserts into various implementations of a concurrent B-tree. Each insertion is performed as a durable transaction,

with a randomly chosen 8-byte integer as the key. In a run with P processes, each process performed either $\lfloor \frac{100000}{P} \rfloor$ or $\lfloor \frac{100000}{P} \rfloor + 1$ insertions.

We ran this test in three different environments: in a normal B$^+$-tree implemented using LibXac, a CO B-tree using LibXac, and on a Berkeley DB [29] B-tree. The block size for both B-trees is 4096 bytes, and the keys of all B-trees are actually padded to 512 bytes.[2]

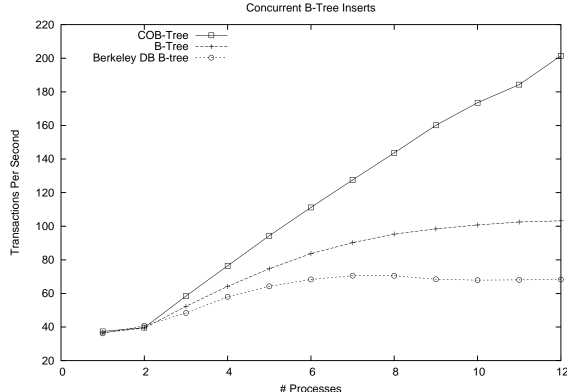

Figure 2: 100,000 Inserts on concurrent B-trees performed by multiple processes.

Figure 2 presents preliminary performance results. Note that the poor performance of the two B-trees relative to the CO B-tree is likely due to the fact that B was not properly tuned. In practice, the effective value of B should be much larger than the default 4K page size specified by Linux.

Since each transaction in this case touches relatively few pages, the cost of a transaction is dominated by the cost of the synchronous disk write during commit. Even though our experiment is run on a machine with only 4 processors and a single disk, all three B-trees are able to achieve speedup using multiple processes because they all implement group commit [11].

We do not interpret these results as evidence that one system (LibXac or Berkeley DB) or data structure (CO B-tree vs. normal B-tree) necessarily outperforms the other. Our claim is only that a system that supports memory-mapped transactions and which is competitive with the traditional alternatives can be feasible to implement in practice.

4. MEGALITHIC TRANSACTIONS

Of the three problems we described for a CO B-tree implemented using transactional memory, the problem of megalithic transactions is the most troublesome. In this section, we describe the CO B-tree data structure of [6] in more detail, explain how update operations give rise to megalithic transactions, and briefly comment on ways to address this issue.

Rather than explaining the entire CO B-tree data structure, we focus on a piece of the data structure, the ***packed memory array (PMA)***, that illustrates the problems we faced. A PMA is an array of size $O(N)$, which dynamically maintains N values in sorted order. The values are kept approximately evenly spaced, with small gaps to allow insertions without having to move too many elements on average. Occasionally a large amount of data needs to be moved, but if the gaps are managed properly [6], the average cost of insertions remains small. The CO B-tree stores its values in a PMA, and uses a static cache-oblivious search tree as an index into the PMA. Thus, an insertion into the CO B-tree involves an insertion into the PMA and an update of the index.

We only sketch the algorithm for insertion into the PMA (see [6] for more details). To insert an element into a PMA, if there is a gap between the inserted element's neighbors, then we insert the element into a gap position. Otherwise, we look for a neighborhood around the insertion point that has low *density*, that is, look for a subarray that is not storing too many data elements. Given a sufficiently sparse neighborhood, we *rebalance* the neighborhood, i.e., space out the the elements evenly. In order to get rebalances to run quickly on average, one must apply stricter density rules for larger neighborhoods. The idea is that because rebalancing a large neighborhood is expensive, after the rebalance we need a greater return, i.e., a sparser final neighborhood. Once the neighborhood corresponding to the *entire* PMA is above the maximum density threshold, we double the size of the array and rebuild the entire PMA, and thus the entire CO B-tree.

For the CO B-tree, updates that must rebalance the entire PMA, or at least a large fraction of it, produce what we call a megalithic transaction. A megalithic transaction is one that modifies a huge amount of state, effectively serializing performance. A megalithic transaction represents the extreme case because it is a large transaction that conflicts with all other transactions. Thus, some contention management strategy [16] is needed to avoid livelock when transactions conflict.

An operation that modifies all of memory does not necessarily cause performance problems in the serial case, but it does cause problems in the parallel case. For the serial case, the CO B-tree we used has good amortized performance. Although the average cost of an update is small, some updates are expensive. In a serial data structure, the cost of the expensive operation can be amortized against previous operations. In the parallel implementation, the expensive updates cause performance problems because they increase the critical path of the program, reducing the average parallelism.

If updates are infrequent compared to searches, their performance impact can be mitigated by using a multiversion concurrency control [8], in which read-only transactions never conflict with other transactions. Our LibXac prototype provides multiversion concurrency control and allows the user to specify special read-only transactions that will always succeed. But although we have not confirmed this fact experimentally, this particular CO B-tree data structure appears to have limitations in the parallelism of its update operations.

The best solution for a megalithic transaction is to eliminate it altogether, through algorithmic cleverness. Ideally, deamortizing the operations on a CO B-tree would make the footprint of every update transaction small, thereby eliminating any megalithic transactions. Another approach may be to find a way to split up the large rebalancing operations of the CO B-tree's PMA into multiple transactions. The worst-case time for a single update may still be large in this case, but that update would at least not block all other concurrent transactions from committing.

Since improving or deamortizing a data structure often make the structure more complicated, transactional memory can help us by simplifying the implementation. In general, programming with a transational memory interface instead of with explicit I/O makes it plausible that a programmer could implement an even more sophisticated data structure such as a ***cache-oblivious lookahead array*** [25] (which provides dramatic performance improvements for workloads with many more insertions than searches) or a ***cache-***

[2] We ran our experiments on a 4-processor Opteron 1.4Ghz 840 with 16 GB of RAM, running Suse Linux 9.1 and the 2.6.5-7.155.29-smp kernel. The system had an IBM Ultrastar 146GB 10,000RPM disk with an ext3 filesystem. We placed the logs and the data on the same disk.

oblivious string B-tree [7] (which can handle huge keys efficiently).

Another approach to working around megalithic transactions is to use some sort of loophole in the strict transactional semantics, such as the release mechanism [16] or so-called open-transactions [24].

5. RELATED AND FUTURE WORK

Using LibXac, we are able to overcome the obstacles of transactional I/O and durability for a concurrent CO B-tree implemented using transactional memory.

Our experimental results suggest that LibXac provides acceptable performance for durable transactions. We would pay a high penalty in performance, however, if we were to use LibXac for the sort of non-durable transactions that many transactional-memory systems provide. The LibXac runtime must handle a `SEGFAULT` and make an `mmap` system call every time a transaction touches a new page. This method for access detection introduces a per-page overhead that on some systems can be 10 μs or more. With operating system support to speed up `SEGFAULT` handlers, or introduction of a system call that reports which pages a transaction has read or written, one might be able to use a system such as LibXac for efficient non-durable transactions.

Without operating system or compiler support, implementing durable transactions on memory-mapped data has been viewed as an open research problem [3, 9]. The problem is that the operating system may write memory-mapped data back to disk at any time. The correctness of a transaction logging scheme usually requires that the data be written back to disk only after the log has been written, but with memory mapping, the operating system may write the data back too soon. A relatively slow implementation of portable transactions for single-level stores that incurs a synchronous disk write after every page accessed by a transaction is described by [21]. Recoverable virtual memory [27] supports durable transactions, but the interface requires programmers to explicitly identify the shared memory being accessed by a transaction. LibXac's approach of remapping pages as they are modified is, to our knowledge, the first portable and efficient solution to durable transactions on memory mapped data.

6. REFERENCES

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pp. 316–327, San Francisco, California, Feb. 2005.

[2] C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, Oct. 2005.

[3] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 96–107, Santa Clara, California, Apr. 1991.

[4] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pp. 560–569, Burlington, VT, Oct. 1996.

[5] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, Feb. 1972.

[6] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *FOCS*, pp. 399–409, 2000.

[7] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string b-trees. In *To appear in PODS'06*, Chicago, Illinois, June 2006.

[8] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transaction on Database Systems (TODS)*, 8(4):465–483, Dec. 1983.

[9] P. A. Buhr and A. K. Goel. uDatabase annotated reference manual, version 1.0. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Sept. 1998. `ftp://plg.uwaterloo.ca/pub/uDatabase/uDatabase.ps.gz`.

[10] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[11] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pp. 1–8, Boston, Massachusetts, 18–21 June 1984.

[12] M. Frigo. *Portable High-Performance Programs*. PhD thesis, MIT EECS, June 1999.

[13] J. Gray. The transaction concept: Virtues and limitations. In *Seventh International Conference of Very Large Data Bases*, pp. 144–154, Sept. 1981.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '2004)*, pp. 102–113, Munich, Germany, 19–23 June 1997.

[15] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pp. 388–402, Anaheim, California, Oct. 2003.

[16] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 92–101, 2003.

[17] M. P. Herlihy and J. Moss. Transactional support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, One Kendall Square, Cambridge, MA 02139, Dec. 1992. `ftp://ftp.cs.umass.edu/pub/osl/papers/crl-92-07.ps.Z`.

[18] Z. Kasheff. Cache-oblivious dynamic search trees. M.eng., Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2004. `http://bradley.csail.mit.edu/papers/Kasheff04`.

[19] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming (LFP)*, pp. 105–112. ACM Press, 1986.

[20] A. McDonald, J. Chung, B. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd Intl. Symposium on Computer Architecture (ISCA)*, Boston, Massachusetts, June 2006.

[21] D. J. McNamee. *Virtual Memory Alternatives for Transaction Buffer Management in a Single-level Store*. PhD thesis, University of Washington, 1996. `http://www.cse.ogi.edu/~dylan/Thesis.pdf`.

[22] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM*, 19(7):395–404, July 1976.

[23] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, Austin, Texas, Feb. 2006.

[24] E. Moss and T. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL 05)*, pp. 39–48, San Diego, California, Oct. 2005.

[25] J. Nelson. External-memory search trees with fast insertions. Master's thesis, Massachusetts Institute of Technology, June 2006.

[26] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[27] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, 1994.

[28] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed*, pp. 204–213, Ottawa, Ontario, Canada, 1995.

[29] Sleepycat Software. The Berkeley database. `http://www.sleepycat.com`, 2005.

[30] J. Sukha. Memory-mapped transactions. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, May 2005.